2015 IEEE Trustcom/BigDataSE/ISPA

Prediction Models for Performance, Power, and Energy Efficiency of Software Executed on Heterogeneous Hardware

Dénes Bán, Rudolf Ferenc, István Siket, and Ákos Kiss
Department of Software Engineering
University of Szeged
Szeged, Hungary
Email: {zealot,ferenc,siket,akiss}@inf.u-szeged.hu

***Abstract*—Heterogeneous environments are becoming commonplace so it is increasingly important to understand *how* and *where* we could execute a given algorithm the most efficiently. In this paper we propose a methodology that uses both static source code metrics and dynamic execution time, power and energy measurements to build configuration prediction models. These models are trained on special benchmarks that have both sequential and parallel implementations and can be executed on various computing elements, e.g., on CPUs or GPUs. After they are built, however, they can be applied to a new system using only the system's static metrics which are much more easily computable than any dynamic measurement. We found that we could predict the optimal execution configuration fairly accurately using static information alone.**

***Keywords*-Green computing, heterogeneous architecture, performance optimization, power-aware execution, configuration selection**

I. Introduction

As technological advancements make GPUs – or other alternative computation units – more widespread, it is increasingly important to question whether the CPU is still the most efficient option for running specific applications. In this document we describe a method for deriving prediction models that can select the execution configuration best suited for a given algorithm with regards to one of three different aspects: time, power, or energy consumption. These models are built by applying various machine learning methods where the predictors are calculated from the source code (using static analysis techniques) and the output of the models is the optimal execution configuration.

To build the desired prediction models, first we take a number of algorithms – referred to as benchmarks – that have functionally equivalent sequential and parallel (OpenCL and OpenMP-based) implementations. After this, we extract multiple size, coupling and complexity metrics from the main functional parts of every analyzed benchmark using static source code analysis. Then we collect measurements on the time and power required to run these algorithms on different platforms and assign labels to them based on which configuration performed the best. Finally, we apply multiple machine learning methods that use the calculated metrics to predict the optimal execution configuration for a system.

These steps yield a model – one for every machine learning approach – that is capable of classifying new systems as well. There are no prerequisites for using the created models other than extracting the same static metrics from the new subject system's source code that were used in the model building phase. With those metrics, one of the previously built models can be utilized to predict the optimal configuration for running the subject system. In this document we describe a possible method for creating such models through a concrete experiment and discuss their benefits as well as possible ways for improving them even further.

The paper is organized as follows: In the next section we list some works related to ours. Then, in Section III we describe our methodology in detail. In Section IV we introduce the used benchmarks, while in Section V we briefly describe the way we performed the dynamic measurements. Afterwards, in Section VI we describe the static metrics extraction method in detail. In Section VII we show the results that we have achieved. Finally, in Section VIII we draw conclusions and outline future work.

II. Related Work

As heterogeneous execution environments grew to be more and more prevalent in recent years, it also became increasingly important to study their individual and relative performances. There is a multitude of related work in the area with fundamentally different approaches.

Some researchers tried to characterize a particular platform alone. For example, Ma et al. [1] focused only on GPUs and built statistical models to predict power consumption. Brandolese et al. [2] concentrated on CPUs by statically analyzing C source code and estimating their execution times. For the OpenMP environment, Li et al. [3] derived a performance model while Shen et al. [4] compared OpenMP to OpenCL using some of the same benchmark systems we used. For FPGAs, Osmulski et al. [5] introduced a tool to evaluate the power consumption of a given circuit without needing to actually test them. It is also evident from these references that most of this type of research targets a single aspect (time or power). We on the other hand, consider multiple platforms and

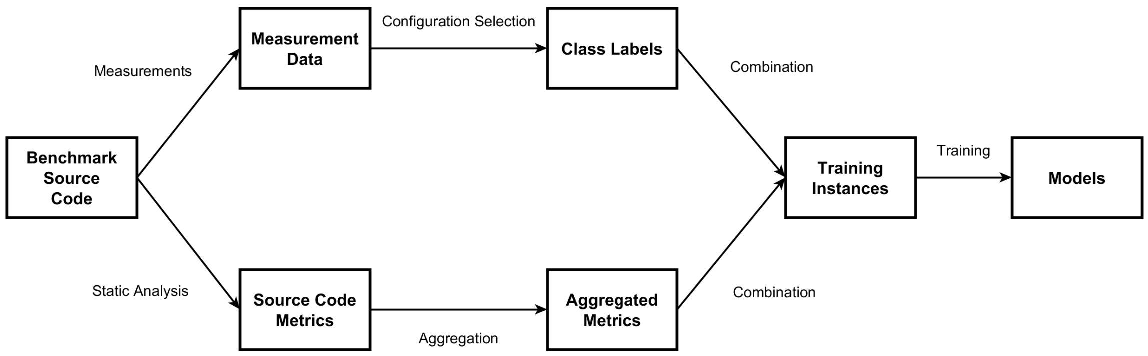

Fig. 1. Main steps of the model creation process

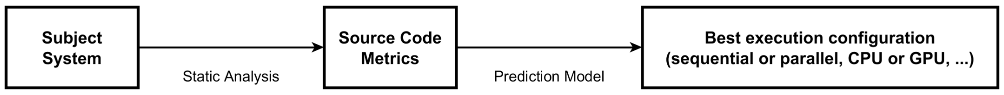
Subject System
Static Analysis
Source Code Metrics
Prediction Model
Best execution configuration (sequential or parallel, CPU or GPU, ...)

Fig. 2. Usage of a previously built model on a new subject system

aspects as our goal is to predict the optimal environment from static information alone.

Others are more closely related to our current work as they focus on cross-platform optimization. Yang et al. [6] generalizes the expected behavior of a program on another platform by extrapolating from partial execution measurements while Takizawa et al. [7] aim at energy efficiency by dynamically selecting the execution environment at run time. Unlike these works, we use dynamic information only for building the prediction models which then can be used with static data alone.

A subset of these cross-platform works concentrate on compiled or intermediate program representations. Kuperberg et al. [8] analyze components and platforms separately to avoid a combinatorial explosion. They build parametric models for performance prediction but it requires microbenchmarks for each platform and works with Java bytecode only. Marin and Mellor-Crummey [9] also process application binaries and build architecture-neutral models which are then used to estimate cache misses and execution time on an unknown platform. One key difference of these studies compared to our approach is that we use the source code of the training benchmarks and not their compiled forms.

III. Methodology

This section contains the detailed description of our concept of a prediction model and how it is built. Using source code metrics produced by static analysis, our model is able to predict the computing unit that allows the fastest or most energy efficient execution of a given program in advance. The model is qualitative, so it does not predict the possible gain of selecting one execution configuration over another, only the best configuration itself. The model is built following these steps:

- Extract multiple size, coupling, and complexity metrics from the main functional parts of the analyzed systems.
- Collect measurements of the time and power required to run them on different platforms.
- Use various machine learning algorithms to build models that are able to predict the optimal configuration for a program with a specific set of metric values.

The steps and intermediate states of our methodology are outlined in Figure 1. Each of these steps will be detailed in their dedicated sections:

- The selected benchmarks in Section IV,
- the dynamic measurements in Section V,
- the static analysis in Section VI-A,
- the selected metrics relevant for representing the encapsulated algorithms in Section VI-B,
- the metric aggregation process and its result in Section VI-C, where a single set of metrics is collected for every benchmark,
- the configuration labeling and the combination of labels and metrics into instances in Section VI-D, and finally,
- the model training and its results in Section VII, where we use `Weka` and a number of its classifier algorithms to build the model we aim for.

Once a prediction model is in place, new systems can be analyzed to predict their optimal execution configuration. Figure 2 depicts the steps of applying a model to a new subject system (unknown to the trained model). To determine the optimal execution configuration of a new system, all we have to do is to calculate the same source code metrics (via static analysis) that we used for training the model, and let the model decide.

IV. Benchmarks

For subject systems to train our models on, we used the systems found in two self-contained benchmark suites: *Parboil* and *Rodinia*. The Parboil [10] suite provides a combination of sequential, OpenCL, and OpenMP implementations for 11 programs. Rodinia [11] contains 18 benchmark programs with OpenCL and OpenMP implementations but without the sequential equivalents. In this work, not all of these programs were measured, either because they had only OpenCL or only OpenMP implementations, but not both, or because their input sets were too complex. Note that during metrics calculation (see Section VI) further systems needed to be skipped either because of a faulty build (inherent include errors) or because a single main file contained the whole logic of the program and therefore it could not be separated from the OpenCL specific overhead, causing large deviations in the computed metrics. The final number of systems that have both metric data and measurements are 7 and 8 for Parboil and Rodinia, respectively.

V. Measurements

In order to train our configuration prediction models, we needed to obtain dynamic measurements for execution time, power consumption and energy usage. We compiled the benchmarks with `g++` 4.8.2 using standard `-fopenmp` or `-lOpenCL` flags and ran them on a platform built from 2 Intel Xeon E5-2695 v2 CPUs (30M Cache, 2.40 GHz), 10 $\times$ 8GB of DDR3 1600 MHz memory, a Supermicro X9DRG-QF mainboard, and an AMD Radeon R9 290X VGA card. Execution time could be easily checked using software-based timers. Power and energy, on the other hand, required a more sophisticated approach.

We applied a universal hardware-extension solution that is completely reversible and yields reliable measurements without affecting performance or stability. We placed high precision resistors into each of the power lines of the relevant processing units and used a multi-channel oscilloscope to measure their voltage drops over time. Because these drops are proportional to the current flowing through the resistors, we could convert the data from the oscilloscope to power and energy figures. A more detailed description of the dynamic measurement is out of the scope of this paper.

VI. Metrics Extraction

In this section we briefly describe the process of static analysis to calculate static source code metrics. As outlined in Section III, this static source code information is used to predict target execution configuration for systems. We list all the selected metrics used in the machine learning algorithms as predictors and present how we aggregated the function level metrics to system level.

A. Static Analysis

For metrics calculation, we ran our static analysis tool on both benchmark suites. This tool [12] analyzes the source code, computes different kinds of metrics and outputs the result into `.csv` (comma separated values) files separately for every type of high-level source code entity – classes, functions, etc. Considering the procedural structure of the benchmark systems, we used function level metrics as the basis for further processing.

Note that the precision of the source code metrics could be improved by using block-level metrics but that would require the manual annotation of every benchmark system. Moreover, the current approach does not use any dynamic information from the source code yet, metrics are static, and do not contemplate run time problems such as caching or memory allocation. That is because dynamic information is much more difficult to collect, but it should be definitely considered for further improvement of the prediction models. As a first step, we believe static information offers a good trade-off between efficient data collection and prediction accuracy.

B. Metric Definitions

The following metrics were computed and used as predictors for the classifications:

- **McCabe's cyclomatic complexity (McCC)**: McCC is defined as the number of decisions within the specified function plus 1, where each *if, for, while, do ... while* and *?: (conditional operator)* counts once, each N-way *switch* counts N+1 times and each *try block* with N *catch blocks* counts N+1 times. (E.g., *else* does not increment the number of decisions.)
- **Nesting level (NL)**: NL for a function is the maximum of the control structure depth. Only *if, switch, for, while* and *do... while* instructions are taken into account.
- **Nesting level else-if (NLE)**: NLE for a function is the maximum of the control structure depth. Only *if, switch, for, while* and *do... while* instructions are taken into account but *if ... else if* does not increase the value.
- **Number of incoming invocations (NII)**: NII for a function is the cardinality of the set of all functions which invoke this function.
- **Number of outgoing invocations (NOI)**: NOI for a function is the cardinality of the set of all function invocations in the function.
- **Logical lines of code (LLOC)**: LLOC is the count of all non-empty, non-comment lines of a function.
- **Number of statements (NOS)**: NOS is the number of statements inside a given function.

Note that all of these metrics can be statically computed. Nevertheless, they can be used to predict dynamic behavior fairly well, as we will demonstrate this in Section VII.

C. Metrics Aggregation

The output of the static analysis is a set of metrics for every function in every implementation variant of every benchmark system. To aggregate these metrics into a system-level set for each benchmark system, first we combined the metrics of multiple functions per benchmark implementation. The method we used for aggregation in the current paper is addition, but future work may experiment with different, potentially more complex functions, perhaps even different ones per metric type. Note that while addition might not always be the best aggregation method for specific metrics (e.g., inheritance depth, or comment density), it is a natural and expressive choice for the metrics we use in this work.

Next, we inspected the differences in the results per implementation variant for a given benchmark system. We noticed that while the sequential and OpenMP variant nearly always yielded the same – or negligibly different – metrics, the OpenCL variant was significantly larger. This turned out to be because:

- the main files (`main.cpp`, `main.cc`, `main.c`) of the OpenCL variants in every benchmark system increased the size and complexity because of the integration characteristics of OpenCL itself (the represented algorithms were not part of the main files),
- the source code of the OpenCL variant frequently contained OpenCL specific headers and files which implemented functionality that the other variants assumed to be implicitly available.

By filtering out these "unnecessary files", the computed metrics "converged" to a single set and this supports that they really only represent the enclosed algorithm. The remaining marginal differences were handled by taking the maximum of the values across the variants.

This way we got one single set of metrics for every benchmark system, capturing its characteristics.

D. *Configuration Selection*

After we have obtained measurements for each aspect (time, power, and energy) in each implementation variant (sequential, OpenMP on CPU and OpenCL on GPU) for each benchmark system, the question is not how fast (or energy efficient) a given algorithm will be, but in which execution configuration will it be the fastest (or most energy efficient). To this end, we assigned three labels to each benchmark system, one for each aspect, denoting the best execution configuration for each aspect. The possible labels are `SEQ-CPU`, `OMP-CPU` and `OCL-GPU` for the CPU-based sequential, CPU-based OpenMP and GPU-based OpenCL configurations, respectively.

The resulting `.csv` files for systems in the two benchmark suites can be seen in Table I and Table II. Note, that while Rodinia (Table II) only has the two possible labels present in its table, Parboil (Table I) could have three labels (SEQ-CPU, OMP-CPU and OCL-GPU), but OMP-CPU is not present there because it is never optimal.

These results were then written into `.arff` files with the last three label columns interpreted as nominal values. The `.arff` format (Attribute-Relation File Format) is the internal data representation format of `Weka` [13]. It is an ASCII text file that describes a list of instances sharing a set of attributes. These attributes can be strings, dates, numerical values and nominal values, the last of which can be used to represent class labels.

The Tables I and II reveal that the optimal configuration from the energy aspect was constant for both benchmark suites and the optimal configuration from power and time aspects were so strongly correlated that they were always identical. Because of this, we chose not to consider energy labels and to merge power and time labels into a single one for further experiments.

VII. Results

In this chapter we describe how and what types of prediction models have been built. We also present the validation results of the models created by different machine learning algorithms. The results are validated with 4-fold cross-validation [14].

A. *Machine Learning*

Using the data shown in Tables I and II, we were able to run various machine learning algorithms to build models that can predict the configuration labels based on the source code metrics. We performed the machine learning with `Weka` [13], using multiple classifier algorithms and the wrapper script shown in Listing 1.

As can be seen, we applied four different algorithms for learning: J48 decision tree, Naive Bayes classifier, Logistic regression and sequential minimal optimization function (SMO).

Listing 1. Machine learning Weka script

```
for BENCH in parboil rodinia
do
  java —cp weka.jar weka.core.converters.CSVLoader —N 8 ↩
       ../java/${BENCH}.csv > ${BENCH}.arff
  touch ${BENCH}.txt
  for CLASSIFIER in trees.J48 bayes.NaiveBayes functions.↩
       Logistic functions.SMO
  do
    for CLASS in 8 # possibly more
    do
      java —cp weka.jar weka.classifiers.${CLASSIFIER} —t↩
           ${BENCH}.arff —c ${CLASS} —i —x 4 >> ${BENCH}.↩
          txt
    done
  done
done
```

B. *Validation of the Models*

Our first experiment was conducted using the J48 decision tree, which is an open-source implementation of the well-known C4.5 algorithm [15]. It produced 100% precision in both cases which is not surprising as there is a clear division between the two possible labels using only a single metric. This means, that we can select a value of a metric so that all the systems having higher metric value than that fall into one class, while systems with lower metric value fall into another class. The learning algorithms can find these values and achieve 100% precision. For Parboil, it is the NOI metric (over value 15 the label is OCL-GPU, otherwise it is SEQ-CPU), and for Rodinia, it is the NII metric (over value 3 the label is OCL-GPU, otherwise it is OMP-CPU). These simple separations are illustrated in Table III and Table IV. Note that for Rodinia, every other metric provides the same linear separation that NII does.

The final decision trees produced by the J48 algorithm for Parboil (left) and Rodinia (right) can be seen in Figure 3.

The Logistic regression model [16] – similarly to the decision tree – is perfectly accurate as there is a clear separation based on numeric predictors as described above.

Benchmark	McCC	NL	NLE	NII	NOI	LLOC	NOS	TimeLabel	PowerLabel	EnergyLabel
Mri-Q	20	6	6	6	17	129	50	OCL-GPU	OCL-GPU	SEQ-CPU
Mri-Gridding	24	11	11	6	6	135	56	SEQ-CPU	SEQ-CPU	SEQ-CPU
Spmv	5	2	2	2	15	48	15	SEQ-CPU	SEQ-CPU	SEQ-CPU
Lbm	59	35	35	19	25	519	135	OCL-GPU	OCL-GPU	SEQ-CPU
Stencil	8	4	4	2	19	60	18	OCL-GPU	OCL-GPU	SEQ-CPU
Histo	13	5	5	3	10	97	33	SEQ-CPU	SEQ-CPU	SEQ-CPU
Cutcp	53	18	18	9	29	340	157	OCL-GPU	OCL-GPU	SEQ-CPU

TABLE I
TRAINING INSTANCES FROM THE PARBOIL SUITE

Benchmark	McCC	NL	NLE	NII	NOI	LLOC	NOS	TimeLabel	PowerLabel	EnergyLabel
Streamcluster	249	66	66	49	160	1263	735	OCL-GPU	OCL-GPU	OMP-CPU
Leukocyte	672	134	134	99	260	2426	1627	OCL-GPU	OCL-GPU	OMP-CPU
Kmeans	100	22	22	9	53	487	240	OCL-GPU	OCL-GPU	OMP-CPU
Nw	21	3	3	3	14	104	58	OMP-CPU	OMP-CPU	OMP-CPU
Bfs	17	5	5	2	13	107	56	OMP-CPU	OMP-CPU	OMP-CPU
Pathfinder	20	6	6	3	10	87	52	OMP-CPU	OMP-CPU	OMP-CPU
Cfd	156	62	54	70	142	1424	776	OCL-GPU	OCL-GPU	OMP-CPU
Lavamd	85	6	6	7	17	370	128	OCL-GPU	OCL-GPU	OMP-CPU

TABLE II
TRAINING INSTANCES FROM THE RODINIA SUITE

McCC	NL	NLE	NII	NOI	LLOC	NOS	Label
53	18	18	9	**29**	340	157	OCL-GPU
59	35	35	19	**25**	519	135	OCL-GPU
8	4	4	2	**19**	60	18	OCL-GPU
20	6	6	6	**17**	129	50	OCL-GPU
5	2	2	2	**15**	48	15	SEQ-CPU
13	5	5	3	**10**	97	33	SEQ-CPU
24	11	11	6	**6**	135	56	SEQ-CPU

TABLE III
CLEAR SEPARATION OF THE PARBOIL BENCHMARK SUITE BY THE NOI METRIC

McCC	NL	NLE	NII	NOI	LLOC	NOS	Label
672	134	134	**99**	260	2426	1627	OCL-GPU
156	62	54	**70**	142	1424	776	OCL-GPU
249	66	66	**49**	160	1263	735	OCL-GPU
100	22	22	**9**	53	487	240	OCL-GPU
85	6	6	**7**	17	370	128	OCL-GPU
21	3	3	**3**	14	104	58	OMP-CPU
20	6	6	**3**	10	87	52	OMP-CPU
17	5	5	**2**	13	107	56	OMP-CPU

TABLE IV
CLEAR SEPARATION OF THE RODINIA BENCHMARK SUITE BY THE NII METRIC

Next, we tried the Naive Bayes classifier that yielded 71.4% precision for Parboil and 100% precision for Rodinia. The confusion matrix for the first case can be seen in Table V. The upper left value shows how many instances were correctly identified as `OCL-GPU` and the upper right value shows the number of `SEQ-CPU` instances that were wrongly classified as `OCL-GPU`. Similarly, the lower right value is the number of correctly classified `SEQ-CPU`s while the lower left is the number of `OCL-GPU`s that were classified as `SEQ-CPU`.

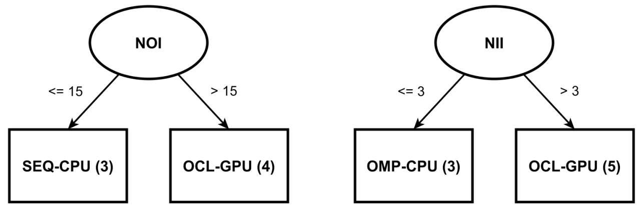

Fig. 3. The final J48 decision trees for Parboil (left) and Rodinia (right)

		Predicted	
		OCL-GPU	SEQ-CPU
Measured	OCL-GPU	2	2
	SEQ-CPU	0	3

TABLE V
THE BAYES CONFUSION MATRIX FOR PARBOIL

Finally, we used a sequential minimal optimization function (SMO). It produced 71.4% and 75% precision for Parboil and Rodinia, respectively. The corresponding confusion matrices can be seen in Table VI (Parboil) and Table VII (Rodinia).

		Predicted	
		OCL-GPU	SEQ-CPU
Measured	OCL-GPU	2	2
	SEQ-CPU	0	3

TABLE VI
THE SMO CONFUSION MATRIX FOR PARBOIL

		Predicted	
		OCL-GPU	OMP-CPU
Measured	OCL-GPU	3	2
	OMP-CPU	0	3

TABLE VII
THE SMO CONFUSION MATRIX FOR RODINIA

For the time being, the results are validated with 4-fold cross-validation [14] inside `Weka`. In a 4-fold cross-validation process, the original data set is randomly partitioned into 4 subsamples, possibly equal in size. Out of the 4 subsamples, 1 subsample is retained as the validation data for testing the model, and the other 3 subsamples are used as training data. The cross-validation process is then repeated 4 times (the number of folds), with each of the 4 subsamples used exactly once as the validation data. The results from the folds are then averaged to produce a single estimation.

Although these findings can hardly be considered widely generalizable due to the small number of instances, the main result of this study is the streamlined process by which they were produced. With the described infrastructure in place, making the model more precise is largely just a matter of integrating more benchmark source code into the analysis.

VIII. Conclusions and Future Work

The goal of this paper was to present our work addressing the creation of prediction models that are able to automatically determine the optimal execution configuration of a program (i.e., sequential, OpenCL, or OpenMP). For this, we developed a highly generalizable and reusable methodology for producing such models. Moreover, these models do not depend on dynamic behavior information so they can be easily applied for classifying new subject systems.

Building these models required a set of algorithms that are each implemented on every relevant target platform. After thorough research, we found two independent benchmark suites containing multiple systems that fulfill this criterion. To be able to build the necessary models, we also needed to measure the time, power, and energy consumption of the algorithms on different configurations. For this, we used a universal solution to measure the power and energy consumption of the hardware components. We then successfully applied our methodology on these systems to create prediction models based on different machine learning approaches, using source code metrics as predictors.

The resulting models are qualitative which means that they can predict the optimal execution configuration, but not how much better it is compared to the other alternatives. Nevertheless, since all the necessary performance information is available, the methodology will be later expanded to produce quantitative models that will make it possible to even estimate the differences.

There are other opportunities for improving the model building process in the future, as well. One of these is increasing the number of instances on which the models are based. Another factor can be the granularity of metric values which can possibly skew because the function-level calculation covered more – or did not cover all – functionality the benchmark systems represent. We tried to mitigate this skewing by filtering unnecessary files but a more general and reliable solution would be block scope-level metric calculation and manual benchmark annotation. A task of that magnitude, however, was outside the scope of this work. We also intend to take platform specific configurations and compiler settings into account.

Overall, we consider the results of this paper encouraging. Despite the small number of subject systems, we were able to demonstrate that statically computed metrics are appropriate and useful for configuration selection. For example, some of the built models reached 100% accuracy in inferring the optimal execution configuration. The models are promising by themselves, but we feel that another main result of this paper is the methodology behind their creation. We now have a flexible, expandable and configurable infrastructure in place and the generalizability of its output models depend only on the number of initial benchmark systems we use for training.

Acknowledgments

The authors would like to thank Péter Molnár and Róbert Sipka for their extensive help with dynamic measurements. The publication is supported by the European Union FP7 project "REPARA – Reengineering and Enabling Performance And poweR of Applications" (project number: 609666).

References

[1] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical power consumption analysis and modeling for gpu-based computing."

[2] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "Source-level execution time estimation of c programs," in *Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on*, 2001, pp. 98–103.

[3] D. Li, B. de Supinski, M. Schulz, K. Cameron, and D. Nikolopoulos, "Hybrid mpi/openmp power-aware computing," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–12.

[4] J. Shen, J. Fang, H. Sips, and A. Varbanescu, "Performance gaps between openmp and opencl for multi-core cpus," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, 2012.

[5] T. Osmulski, J. T. Muehring, B. Veale, J. M. West, H. Li, S. Vanichayobon, S.-H. Ko, J. K. Antonio, and S. K. Dhall, "A probabilistic power prediction tool for the xilinx 4000-series fpga," in *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, ser. IPDPS '00. London, UK, UK: Springer-Verlag, 2000, pp. 776–783. [Online]. Available: http://dl.acm.org/citation.cfm?id=645612.663302

[6] L. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, Nov 2005.

[7] H. Takizawa, K. Sato, and H. Kobayashi, "Sprat: Runtime processor selection for energy-aware computing," in *Cluster Computing, 2008 IEEE International Conference on*, Sept 2008, pp. 386–393.

[8] M. Kuperberg, K. Krogmann, and R. Reussner, "Performance prediction for black-box components using reengineered parametric behaviour models," in *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, ser. CBSE '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 48–63. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87891-9_4

[9] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '04/Performance '04. New York, NY, USA: ACM, 2004, pp. 2–13. [Online]. Available: http://doi.acm.org/10.1145/1005686.1005691

[10] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois, at Urbana-Champaign, Tech. Rep., 2012.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[12] Rudolf Ferenc et al., *REPARA deliverable D2.2: Static analysis techniques for AIR generation*, 2014. [Online]. Available: http://repara-project.eu/

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," in *SIGKDD Explorations*, vol. 11. ACM, Jun. 2009, pp. 10–18.

[14] S. Arlot and A. Celisse, "A survey of cross-validation procedures for model selection," in *Statistics Surveys*, vol. 4, 2010, pp. 40–79.

[15] J. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[16] D. Hosmer and S. Lemeshow, *Applied Logistic Regression*. Wiley-Interscience, 1989.